Apple III Pascal Update

An Introduction to Version 1.1 of Apple Pascal for Improved Application Program Development

Acknowledgements

The Apple III Pascal system is based on UCSD Pascal. "UCSD PASCAL" is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only and is an indication that the associated product or service has met quality assurance standards prescribed by the University. Any unauthorized use thereof is contrary to the laws of the State of California.

Contents

1

Introduction to Apple Pascal Version 1.1

1
Introduction to Apple Pascal Version 1.1

This chapter explains the software changes that came with your Version 1.1 of Pascal. It tells you what to do with your Pascal 1.Ø system files and points out alternate ways of arranging your system program files on disk.

Occasionally in this manual you will see an indented paragraph preceded by one of these symbols:

This means the indented paragraph contains information that will give you a helping hand.

This tells you to be alert. The indented paragraph describes a special or unusual aspect of Apple III Pascal.

How to Use This Manual

The document you are reading is an update of the four Apple III Pascal manuals packaged with your Pascal system software disks. The changes described in this update are additions made to the Pascal system since the production of Version 1.Ø. In general, the execution of your program files made with Pascal 1.Ø will not be affected by the upgrade of your system files to Pascal 1.1. You may enhance your old program files by revising them to incorporate some of the new features of Apple III Pascal presented in the following chapters.

Two types of changes have been made:

- features of interest to all users of Apple III Pascal, and
- features of special interest to programmers who write application programs using numerous interrelated files.

The topics of general interest can be found mainly in Chapters 7, 8, and 9 of this manual. Those of special interest to application

"Using the Extended Libraries." Scan Table 1-1 to find out what topics might be related to your own needs and then study the appropriate sections. Chapter 5 contains no new features of Pascal 1.1, but illustrates the use of many of the new features by means of a sample, complex application.

You may want to go back to the pages in the original Apple III Pascal manual where a particular topic was introduced. For example, if you are not currently knowledgeable about the Pascal system prefix, you may want to review the pages discussing the prefix in the Apple III Pascal Introduction, Filer, and Editor manual. At the same time, you should study the chapter in this manual on new ways of handling the system prefix and program pathnames.

New or Improved Feature	Advantage to Your Program Development	Chapter Reference
AIIFORMAT	Simplifies creating Apple II Pascal disks for the Apple II or the Apple III.	9
CHAINING PROGRAMS	Retains console display mode and contents during the chain between programs.	9
CHAINSTUFF PROCEDURES		3
SET_PREFIX	Sets Pascal file name prefix from your program.	
GET_PREFIX	Returns current prefix value to your program.	
GET_PATHNAME	Returns pathname of program currently executing.	
COMPILE-LISTING FILE REQUEST	Allows you to specify listing file name when starting up the compiler.	7
COMPILER MODIFICATION	Compiler and linker now support 254 procedures --also larger ones--in a single compiled unit.	9
EDITOR EXTENSION	Displays control characters on screen.	9
EXTENDED LIBRARY FACILITY	Allows up to five runtime program libraries. Allows several programs to share library files. Eliminates dependency on SYSTEM.LIBRARY for runtime programs.	2
EXTENSION TO THE FILE NAME SYNTAX	Specifies the directory or subdirectory that contains the currently executing program.	4
FLOATING-POINT I/O CONVERSIONS	Converts ASCII strings to binary floating-point and vice versa.	6
LARGE DISK SUPPORT	Uses PMOVE, a program to configure your large disk so that it functions as the Pascal system disk.	8
QUITTING THE PASCAL SYSTEM	A new command lets you quit the Pascal system to boot another system or application.	9

Table 1-1. Improvements Available With Apple III Pascal, Version 1.1

How to Use Pascal 1.1 on Your Apple III

This section introduces you to the new Pascal: the hardware and software system requirements, definitions of startup disk and system disk, and steps for converting your system to the new version.

Checking Your System Requirements: Hardware and Software

Pascal 1.1 will run on an Apple III computer with at least 128K bytes of internal memory and one external disk drive in addition to the built-in drive. The external disk drive can be a rigid-disk device, such as the Apple ProFile. See your Apple III Standard Device Drivers Manual and the instructions that came with your particular disk drive.

Your Pascal 1.1 software package includes four disks: three contain the Pascal system files, and one is an unformatted (blank) disk to use for developing programs or storing files. The arrangement of the system files on the three new disks is identical to the arrangement on the three disks of Version 1.Ø, with these exceptions: the PMOVE.CODE file is new with Pascal 1.1, and the SOS.DRIVER and SYSTEM.LIBRARY files on the PASCAL3 disk are expanded versions of their counterparts on the PASCAL1 disk. Chapter 8 shows their differences. Table 1-2 shows the way your files are arranged on the new disks. Of course the order of files on each disk doesn't matter.

PASCAL1 (boot and system disk)	PASCAL2 (text and language processing disk)	PASCAL3 (Pascal utilities disk)
SOS.KERNEL	SYSTEM.EDITOR	LIBMAP.CODE
SOS.DRIVER	SYSTEM.SYNTAX	LIBRARY.CODE
SOS.INTERP	SYSTEM.COMPILER	SETUP.CODE
SYSTEM.PASCAL	SYSTEM.ASSMBLER	AIIFORMAT.CODE
SYSTEM.MISCINFO	OPCODES.65Ø2	SOS.DRIVER
SYSTEM.LIBRARY	ERRORS.65Ø2	SYSTEM.LIBRARY
SYSTEM.FILER	SYSTEM.LINKER	PMOVE.CODE

Table 1-2. The Arrangement of Pascal 1.1 System Files As Shipped

Throughout this manual you will see the terms boot disk and system disk. A boot or startup disk is a disk containing the software programs necessary to start up your Apple III. It must contain at least these three files, in any order:

SOS.KERNEL	(the Apple III operating system)
SOS.DRIVER	(the drivers that communicate with your devices)
SOS.INTERP	(the Pascal interpreter)

A startup disk must be in the built-in drive when you turn the power on or press CONTROL-RESET.

A system disk is a disk containing the software necessary to operate a particular language system, such as Apple III Pascal 1.1, or an application. A Pascal system disk must contain at least these two files:

SYSTEM.PASCAL	(the Pascal operating system)
SYSTEM.MISCINFO	(the Pascal configuration information)

To complete the process of bringing up the Pascal system or a Pascal-based application, you must be sure that the system disk is in the system drive (unit #4), whether you are using a flexible or rigid-disk device for file storage. Of course your system disk may also include a SYSTEM.LIBRARY file and a SYSTEM.STARTUP file, along with others from Pascal 1.1, if you need them for program development or for a runtime application.

The startup disk files and the system disk files may be combined on one disk. This is the case with the PASCAL1 disk shipped to you and shown in Table 1-2. For the development or running of any application, you have the option of combining the necessary startup and system files or, in cases where it is more convenient, of starting up in separate stages, with the startup and system files on separate disks.

Although the typical Pascal system uses the Apple III built-in drive for the system disk drive (unit #4), in Chapter 8 you will learn how to use PMOVE to designate any disk drive as the system disk device (including ProFile or any other rigid-disk drive), keeping the built-in drive as the startup drive.

Upgrading Your Apple III Pascal System Software

To upgrade your Pascal system to Version 1.1, follow these three steps:

1. Copy the New Disks

   Be sure to make a copy of each of the three new Pascal 1.1 disks (PASCAL1, PASCAL2, and PASCAL3) and to store the originals as backups. To do this, use the Pascal Filer or the Copy Volume command on your System Utilities disk, as described in your Apple III Owner's Guide.

2. Next, take any Apple III Pascal 1.Ø originals, with their copies, out of your disk file box and replace them (originals and copies) with the new Pascal 1.1 disks.

DO NOT USE ANY OF THE SYSTEM FILES FROM THE OLD VERSION.

Because almost all of the files have been modified in the new version, the old ones no longer serve a useful purpose. Discarding or storing the old version will prevent hazardous mixups in the future. The new files support the same program tools that were available under the old version, including all the necessary SOS, Pascal, Library, and program development files like the Compiler, Linker, and Assembler.

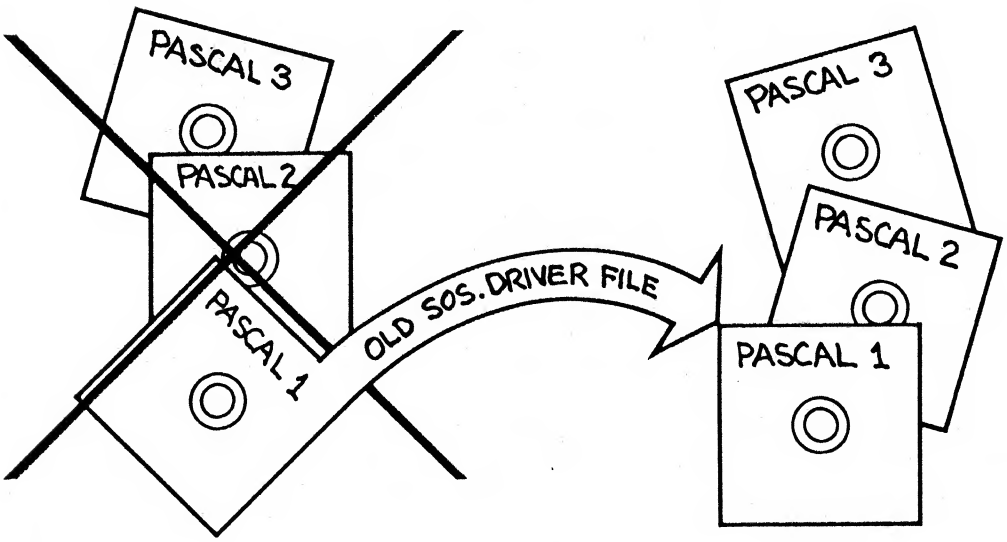


Figure 1-1. Replacing Your Old System Files

The SOS.DRIVER file is the only one you might want to save from your old PASCAL1 disk--or whatever you have been using for your Pascal startup disk. Because the old SOS.DRIVER file stores the startup configuration you already tailored for your particular system, you can save yourself the trouble of reconfiguring the system by transferring your old SOS.DRIVER file to your new copy (your copy, not the original) of the Pascal startup disk, using the Filer or the Copy File command on your System Utilities disk. As you do this, the copy program will ask you if you want to delete the SOS.DRIVER file already on the new disk. After making sure that you are copying the correct SOS.DRIVER file from your old startup disk, type Y for "Yes."

There are two versions of the SOS.DRIVER file on your Pascal disks, one on the PASCAL1 disk and one on the PASCAL3 disk. Do not remove the expanded version on PASCAL3, because it contains

Apple's rigid-disk device called ProFile and four format drivers for the SOS Utility Filer.

You might want to configure the SOS.DRIVER on your new PASCAL3 disk rather than salvage your old one. In this case, follow the instructions in your Apple III Owner's Guide. Note the warning there to build, store, and test a new configuration on a backup startup disk, to make sure it works correctly before placing it on the startup disk you use regularly.

As soon as your SOS.DRIVER is working correctly, store your old Pascal 1.Ø disks and their copies.

3. Revise your program libraries. The program libraries in applications you developed using Version 1.Ø may have to be revised. In Pascal 1.1 certain units in SYSTEM.LIBRARY--namely, PASCALIO, CHAINSTUFF, REALMODES, and TRANSCEND--have been modified. To use the improved units you will have to update your program libraries to contain the new versions.

2

Using the Extended Libraries

2
Using the Extended Libraries

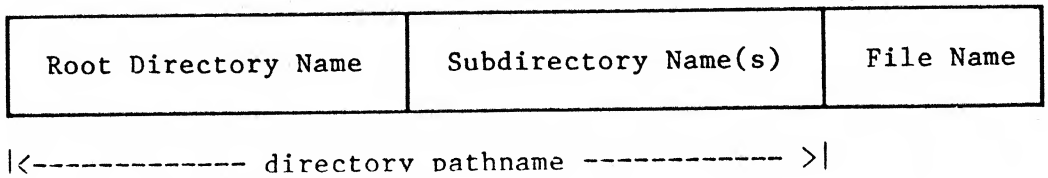
This chapter explains the nature and advantages of the extended library file options available under Pascal 1.1 and illustrates the implementation of library files using the Library Name File, including how programs may share the same library files. At the end, to help you choose your approach to library files as you develop an application, you'll find the sequence of steps followed by the system as it searches library files for the intrinsic units required by a program at execution time.

An Overview of Pascal Libraries

While maintaining the library file system available under Pascal 1.Ø, Pascal 1.1 has increased the number and manageability of library files that you might use for an application.

Important Definitions

In the following discussion, you'll encounter the phrase "in the same directory as." This phrase means that a file must have the same directory pathname as another. Likewise, the phrase "in a different directory than" means that the directory pathnames are different. These definitions apply whether a pathname consists simply of a root directory name and a local file name or whether it consists of a root directory name, one or more subdirectory names, and a local file name. Where there is no subdirectory, the volume root directory name and the directory pathname are the same. Where there is one subdirectory (as in the illustration below), or more than one, the directory pathname includes the root directory and the one or more subdirectories:



In this set of files:

```
/PROFILE              {volume root directory name}
  /SHRLIB             {subdirectory name}
    LIB1.LIB          {a library file}
    LIB2.LIB          {a library file}
```

the directory pathname for the two library files is the same:

```
/PROFILE/SHRLIB/LIB1.LIB
/PROFILE/SHRLIB/LIB2.LIB

|<--directory-->|
     pathname
```

In the case of more than one subdirectory, such as in this set of files:

```
/PROFILE              {volume root directory name}
  /PROGRAM            {subdirectory name}
    PROG1.CODE        {main program code file name}
    PROG1.LIB         {Library Name File}
    LIB1.LIB          {a library file}
    LIB2.LIB          {a library file}
  /LIBRARY            {subdirectory name}
    MYLIB.LIB         {a library file}
    SYSTEM.LIB        {a library file}
```

the directory pathname for MYLIB.LIB and for SYSTEM.LIB is the same:

```
/PROFILE/LIBRARY/MYLIB.LIB
/PROFILE/LIBRARY/SYSTEM.LIB

|<--directory pathname->|
```

But the directory pathname for LIB1.LIB is different:

```
/PROFILE/PROGRAM/LIB1.LIB

|<--directory-->|
     pathname
```

And so we say that SYSTEM.LIB is in the same directory as MYLIB.LIB. Likewise, we say that MYLIB.LIB is in a different directory than LIB1.LIB or LIB2.LIB.

An understanding of Pascal libraries depends on a clear conception of a few other basic terms used frequently in this discussion: application, executable code file, main program, intrinsic unit, and library file.

An application is an executable code file and any associated library

both) have been linked and any required intrinsic units are available in the appropriate library files. An executable code file or a Pascal library file may be composed of different combinations of compiled and assembled source programs.

A Pascal source program called the main program, or sometimes the host program, is the core of the application. The main program is the code that lies at the program level in the Pascal source, as shown in this example:

```
PROGRAM foo;

   {main program declarations}

BEGIN
   {main program code}
END.
```

With Apple III Pascal a main program may use additional code that lies outside its own source. This required code may be in one or more of these sources:

- one or more external assembler routines;
- a set of regular units that must be linked to the main program during program development;
- a set of intrinsic units that are dynamically connected to the main program by the system at execution time.

In this chapter, we refer mostly to intrinsic units, occasionally to regular units. Regular units, by definition, have to be linked with and thereby inserted in the main program code file prior to program execution. Intrinsic units, on the other hand, are connected by the system to the main program at program execution time. Intrinsic units have two characteristics that are relevant to this discussion: first, they are not restricted to use by only one executable program, and second, they must be placed in a library accessible to the system at program execution time in order to be linked to the main program.

Units of either type, regular or intrinsic, are stored in library files. A library file is a code file that is not directly executed. Instead, a library file contains one or more compiled units used by one or more programs. A USES declaration in the program names the required unit, which is connected by the system at program execution time. Another section of this chapter will explain how the system searches various library files for the intrinsic units required by a particular program.

Two or more library files can be combined into one, using a new name or one of the old ones, and units can be moved from one library to another. You may also move units in and out of a copy of the SYSTEM.LIBRARY file that was shipped with your Pascal system. (How to

program, is explained in the Apple III Pascal Program Preparation Tools manual.) The name you give a library file depends on the kind of library file you are using and its purpose. In general, the suffix .LIB is used to complete the file name.

Comparing Libraries Under the Two Versions of Pascal

Note the differences between the library file system supported by Pascal 1.Ø and that supported by Pascal 1.1. For storing units, Pascal 1.Ø allowed only two library files for each executable program: a Program Library File and SYSTEM.LIBRARY. Pascal 1.1 supports these two types and others.

A Program Library File is a library file that is in the same directory as the main program code file and is given the same name as the main program code file except that its suffix is .LIB rather than .CODE. For example, if a main program code file has this directory pathname and file name,

```
/ORT/FOO.CODE
```

then the corresponding Program Library File will have this designation:

```
/ORT/FOO.LIB
```

A Program Library File, like any other library file, may hold between one and sixteen units.

SYSTEM.LIBRARY is a library file that must reside on the system disk in order to be used. It may contain units supplied by Apple Computer, Inc.--the unit called APPLESTUFF, for example--and, if you so choose, additional units that you yourself place in SYSTEM.LIBRARY using the LIBRARY utility program.

The Program Library File and SYSTEM.LIBRARY can both be used by a main program.

In contrast to Pascal 1.Ø, Version 1.1 allows up to six library files (including SYSTEM.LIBRARY) with each main program and also allows multiple programs to share library files. This extension of Pascal libraries is made possible by means of new kind of file, called a Library Name File.

A Library Name File is an Ascii file that you create using the Pascal Editor. In this file, you list the pathnames of up to five library files that contain intrinsic units you want to be used by a main program. As long as its pathname is correctly given, a library file listed in a Library Name File can be in any directory or subdirectory on line at the start of program execution. The Library Name File uses

the same naming convention as a Program Library File: you give it the name of the main program, using .LIB as the suffix. (The specific format for a Library Name File is described in the next section of this chapter.)

Note that if you decide to use a Library Name File, you cannot then use a Program Library File.

By listing library file pathnames in a Library Name File, you direct the system at the start of execution time to search the files with these pathnames to find and link any needed intrinsic units to the main program. Later in this chapter, you'll see how library files in the same directory as the main program or in another directory can be listed in a Library Name File and how they can be shared by multiple application programs.

To be sure, you can use a Program Library File, with or without SYSTEM.LIBRARY, under Pascal 1.1 as well as under Pascal 1.Ø. For a small application, one requiring only a few units, you'll find that a Program Library File will take care of your library file needs. For a larger and more complex application, one using a very large number of code units, you should use instead a Library Name File. Using a Program Library File limits you to units residing in the same directory as the executing program. SYSTEM.LIBRARY also has a limited utility for large applications: it must reside on the system disk, where it takes up valuable space. Furthermore, because you may use a different SYSTEM.LIBRARY in different applications, you face the potential conflict of library units having the same name.

These are the advantages of using Library Name Files for your application programs:

- Up to six library files (including SYSTEM.LIBRARY) can be made available to an executable program. As before, each library file can hold up to sixteen units.

- A library file can be shared by two or more executable programs by listing it in a Library Name File for each one of the executable programs.

- Disk space can be conserved by having only one copy of the same intrinsic unit shared between programs.

Figure 2-1 compares the kinds of library options available in Pascal 1.Ø with those in Pascal 1.1.

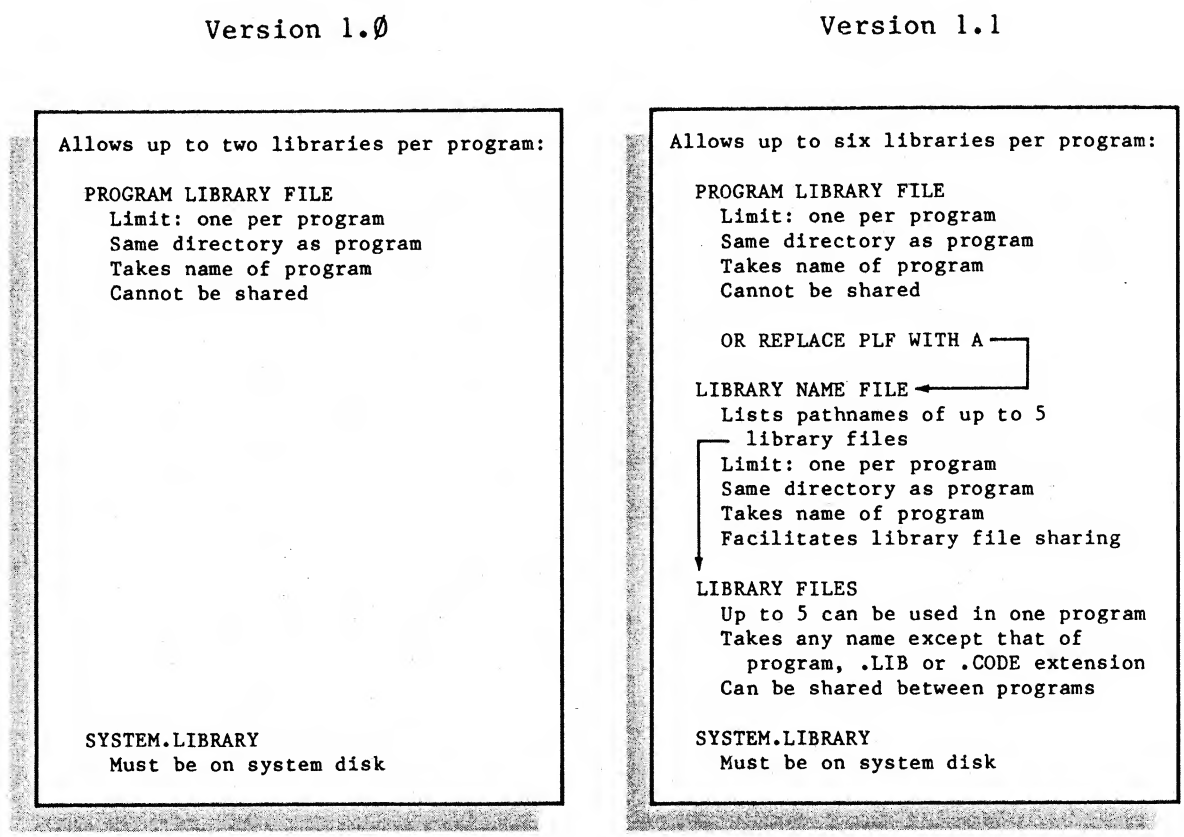


Figure 2-1. Pascal Library Options: Old and New

For information on arranging regular and intrinsic units in libraries, see Chapter 14 in the Apple III Pascal Programmer's Manual, Volume 1. The Apple III Pascal Technical Reference Manual contains useful information on code file formats.

Making a Library Name File

A Library Name File is an Ascii file that must conform to a specific text format.

To make a Library Name File, begin a new file in the Pascal Editor, then type S and then E to bring up on the screen the Set Environment option. Now select the Ascii file option and set it to True by typing A and then T. Exit the Set Environment option by pressing CONTROL-C. Without leaving the Editor, type I to select the Insert option, and make a file using the following format (the general format is on the left; an example file is on the right):

```
LIBRARY FILES:[RETURN]
<pathname>[RETURN]
      .
      .
      .
<pathname>[RETURN]
$$[RETURN]
[CONTROL-C]
```

```
LIBRARY FILES:
/SPLAT/APP1/LIB1.LIB
/SPLAT/APP1/LIB2.LIB
/SPLAT/APP1/LIB3.CODE
$$
```

Notes:

1. The "L" in "LIBRARY" must be the first character on the first line in the file. You cannot have any blank lines, spaces, or other characters at the top of the file or between lines. The string "LIBRARY FILES:" may be uppercase or lowercase characters. Press the RETURN key after each line, as shown.

2. On separate lines, type the pathname followed by RETURN for each file you want to designate as a library file. A pathname can be any legal pathname, up to 80 characters long. You can have up to five pathnames in your file. The system will ignore any pathnames listed after the fifth one.

3. Two dollar signs ($$) make up the last line of the file no matter how many pathnames you use.

4. Press CONTROL-C to leave Insert mode.

After you've made your Library Name File and checked the format carefully, you can type Q, then W, to Write it from the Editor to your program disk, giving it the name of the main program code file but with the .LIB suffix. The following paragraphs tell you in more detail how to select and arrange library files, including those to be shared by using the Library Name Files.

Using Library Files in the Library Name File

This section gives several examples of how to use library files with the Library Name File.

Using One Library File With Two Programs

Suppose you had written two applications, called FOO and GORN, and had stored them in two different directories. Each needs to have a set of intrinsic units on line when being executed. Right now the intrinsics are stored in the library file named BAZ.LIB in the same directory as GORN:

```
/APP                    {a directory}
   GORN.CODE            {an executable program}
   BAZ.LIB              {a library file}
```

Of course, if you wanted either one of the two programs to use the intrinsic units contained in BAZ.LIB, you would have to list the pathname of BAZ.LIB in a Library Name File in the relevant directory, as here:

```
/APP                    {a directory}
   GORN.CODE            {an executable program}
   BAZ.LIB              {a library file}
   GORN.LIB             {a Library Name File . . . . LIBRARY FILES:
                                                      /APP/BAZ.LIB
                                                            .
                                                            .
                                                      $$          }
```

Notice, however, that for both programs to share the same library file--in this case BAZ.LIB--you don't need to place BAZ.LIB in both directories. Instead, you leave the file in the directory /APP and list its pathname in a Library Name File in the other directory, /ORT:

```
/ORT                    {a directory}
   FOO.CODE             {an executable program}
   FOO.LIB              {a Library Name File . . . . LIBRARY FILES:
                                                      /APP/BAZ.LIB
                                                            .
                                                            .
                                                      $$          }
```

Now BAZ.LIB is a shared library file, its intrinsics usable by both programs even though it resides in only one of the two directories.

Using Several Library Files With One Program

If you have a number of library files in the same directory as the executing program, where, for example, the program BAZ.CODE has the pathname

```
/SPLAT/APP1/BAZ.CODE
```

your library files would be placed in the directory

```
/SPLAT/APP1
```

and your Library Name File (with the pathname /SPLAT/APP1/BAZ.LIB) would contain

```
LIBRARY FILES:
/SPLAT/APP1/LIB1.LIB
/SPLAT/APP1/LIB2.LIB
/SPLAT/APP1/LIB3.CODE
$$
```

(LIB1.LIB, LIB2.LIB, and LIB3.CODE are sample names for library files. You may use any name for a file containing library units, as we did for LIB3.CODE, although using .LIB makes it easier to remember that it is a library of units.)

However, if, using the Filer, you set the system prefix to the directory pathname (/SPLAT/APP1) before executing BAZ.CODE, you could write the Library Name File more simply, like this:

```
LIBRARY FILES:
LIB1.LIB
LIB2.LIB
LIB3.LIB
$$
```

The system attaches the prefix to a library file name before opening that file.

When you create shared libraries using Library Name Files, you should be sure that the pathnames for the files to be shared will be correct at execution time. If your executable programs are in different directories or the library files are in different directories, you may need to change the system prefix prior to the execution of each program if you plan to use the prefix in conjunction with the pathnames listed in the Library Name File. For information on a new way to manipulate pathnames during actual program execution, see Chapters 4 and 5.

Using Three Subdirectories

Illustrating the positioning of library files in subdirectories, the following example shows two main programs (PROG1 and PROG2) of an application, each in its own subdirectory, which share some library files but also have library files unique to each program.

Here is Program #1:

```
/PROFILE              {volume root directory name}
   /APPSUB1           {subdirectory name}
      PROG1.CODE      {an executable program}
      UNIT1.LIB       {a library file--not shared between programs}
      UNIT2.LIB       {a library file--not shared between programs}
      PROG1.LIB       {a Library Name File with the following text:

                                          LIBRARY FILES:
                                          APPSUB1/UNIT1.LIB
                                          APPSUB1/UNIT2.LIB
                                          SHRLIB/LIB1.LIB
                                          SHRLIB/LIB2.LIB
                                          $$                  }
```

Here is Program #2:

```
/PROFILE              {volume root directory name}
   /APPSUB2           {subdirectory name}
      PROG2.CODE      {another executable program}
      UNIT3.LIB       {a library file--not shared between programs}
      PROG2.LIB       {a Library Name File with the following text:

                                          LIBRARY FILES:
                                          APPSUB2/UNIT3.LIB
                                          SHRLIB/LIB1.LIB
                                          SHRLIB/LIB2.LIB
                                          $$                  }
```

Finally, here is a subdirectory containing two required library files:

```
/PROFILE              {volume root directory name}
   /SHRLIB            {subdirectory name}
      LIB1.LIB        {a shared library file}
      LIB2.LIB        {a shared library file}
```

Before executing either of the two main programs, you must set the system prefix to /PROFILE.

As each program is executed, the system finds and opens the two library files that are shared, as well as any other specified library files, and uses the intrinsic units stored there. Figure 2-2 illustrates the relationships of the files in the three subdirectories.

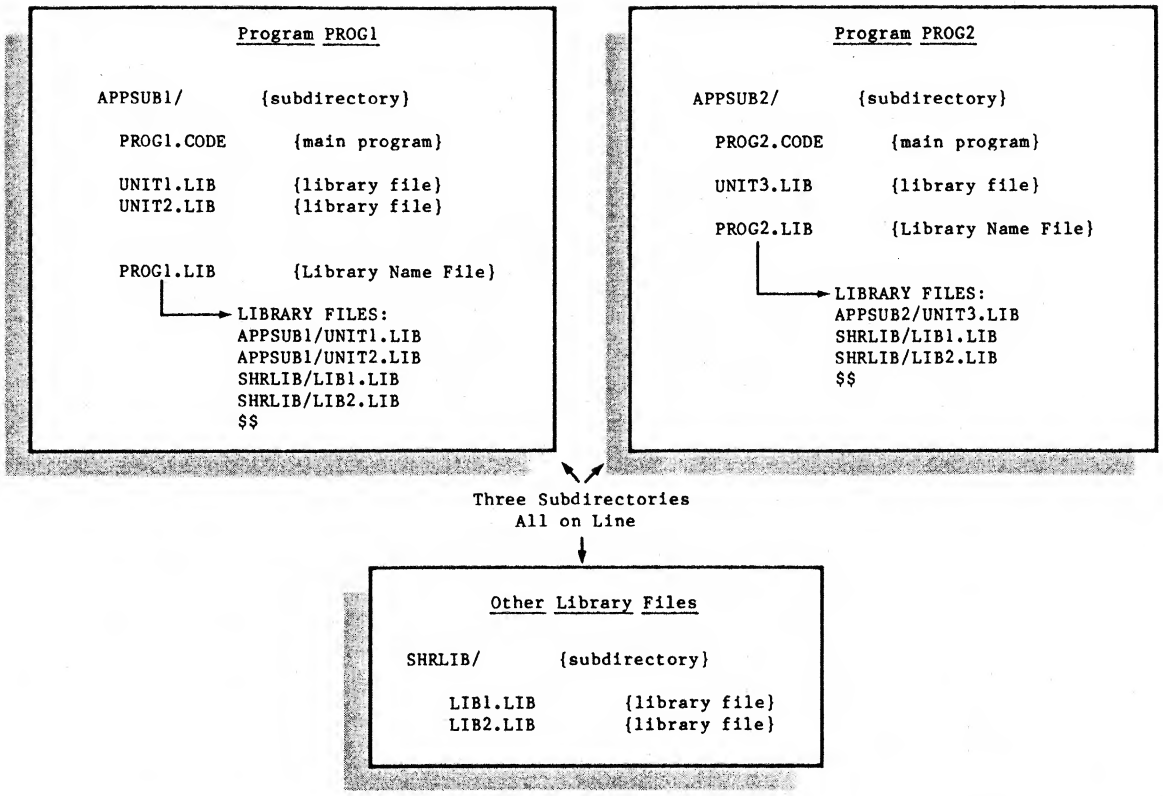


Figure 2-2. Designating Shared Libraries

Note that the library files LIB1.LIB and LIB2.LIB, located in the third subdirectory, are shared by both programs but that UNIT1.LIB, UNIT2.LIB, and UNIT3.LIB are not shared because their pathnames are not listed in the Library Name File of the other program. Had the pathname APPSUB2/UNIT3.LIB been listed in the Library Name File PROG1.LIB, then APPSUB2/UNIT3.LIB would also have become a shared library, usable as well by PROG1 even though the file is physically located in the subdirectory of PROG2.

There are many possible arrangements for program library files, using different combinations of files, directories, and programs. The examples above are simply suggestions and hints to help you get started in developing your own shared libraries. As you can see, you'll want to give considerable thought to the overall structure of your application: the kind of library files appropriate to each program, which files to designate as shared libraries, and the best arrangement on disk of all the files for a particular application program. For an introduction to how to structure an application on disk, see Chapter 5. The next section of this chapter gives a brief description of how libraries are searched for the intrinsic code units required by the executing program.

How the System Searches Libraries

The following step-by-step description will help you choose the library file approach best suited to the particular application you are developing.

When a program is executed, the system first examines it to determine whether or not it uses any intrinsic units. If it does not, the program is loaded and run. If it does, the system searches the different types of library files, in the following order, to find the required units:

1. Program Library File (if the program uses one)
2. Library Name File (if there is no Program Library File)
3. Library files whose pathnames are listed in a Library Name File
4. SYSTEM.LIBRARY (if it is on the system disk)

The system first looks for a file of the same name as the executing program but with the suffix changed from .CODE to .LIB. Then it tries to open the file corresponding to its new name (PROGNAME.LIB). If the file exists, the system determines whether it is a code file or an Ascii file. If it finds a code file (the file we call a Program Library File), the system looks in the file for the required intrinsic units. If it finds instead an Ascii file (the file we call a Library Name File), the system collects the pathnames of the library files listed there and then looks in those files for the required intrinsics.

If you have set a prefix and if the names of the files listed in the Library Name File require a prefix, the system attaches the prefix before searching for the files. (See Chapter 4 for a new extension to pathnames that makes program prefixes available at execution time.)

If there are intrinsics needed that have not been found in a Program Library File or by means of a Library Name File, or if you have not used either of these libraries at all, the system looks in SYSTEM.LIBRARY. If the missing intrinsics are not found in SYSTEM.LIBRARY or if SYSTEM.LIBRARY is not on the system disk, an error message appears on the screen and the system returns control to the Command line.

The system searches for the intrinsic units until it finds them or until it runs out of library files and gives an error message. If it finds the intrinsics before it has looked in all the relevant library files, it stops searching and begins executing the program.

3

Using the Pascal System Prefix at Execution Time

3

Using the Pascal System Prefix at Execution Time

Three new procedures have been added to the CHAINSTUFF unit to give an executing program these capabilities:

- to set or reset the current Pascal system prefix;
- to get the current Pascal system prefix value;
- to get or find out its own pathname.

Setting or getting the prefix this way is equivalent to setting or getting the prefix by using the Filer, except that now you can do any of these procedures from your program while it is executing.

Note that references here are to the Pascal system prefix, not to the SOS prefix.

In order to use one of these procedures, your program must use the intrinsic unit CHAINSTUFF, available in SYSTEM.LIBRARY as part of the original Pascal software.

Setting the Prefix From Your Program

The format for the Set Prefix function call is

```
PREFIX_SET := SET_PREFIX(NEW_PREFIX)
```

where NEW_PREFIX is a parameter of type string that holds the new value for the system prefix and where PREFIX_SET is a user-defined boolean variable. The calling program must ensure that the new prefix is a valid SOS pathname. Any trailing delimiter (/) at the end of the string is removed by the procedure. If for any reason the prefix string is incorrect (is not a valid SOS pathname), the function returns "False" and the prefix keeps its original value. If the syntax of the string is correct, the function returns "True" and sets the system

Here are a couple of reminders about using the system prefix. First, when your system is booted, the prefix automatically will be set to the name of the Pascal system volume being used. That will remain the prefix until you change it, using the Filer or using the new Set Prefix procedure.

Remember also that even after you have set the prefix in order to get to certain files, you can still get to any other file by supplying its full pathname. In other words, your program does not need to reset the prefix every time it refers to a file that is unrelated to the current prefix. The full pathname can be used instead.

To review the Pascal system prefix feature, see the Apple III Pascal Introduction, Filer, and Editor manual, Chapter 3.

Getting the Value of the Current Prefix

The format for the Get Prefix procedure call is

```
GET_PREFIX(CURRENT_PREFIX)
```

where CURRENT_PREFIX is a VAR parameter of type string. The variable passed to this procedure is set to the current value of the system prefix. The prefix has a trailing delimiter (/).

Here is an example:

```
LOCAL_FILE := 'FOO.DATA';
GET_PREFIX(CURRENT_PREFIX);
FNAME := CONCAT(CURRENT_PREFIX, LOCAL_FILE);
```

This example first uses the Get Prefix procedure within an executing program to find out the value of the current system prefix, and then--by means of the CONCAT function--constructs (in the third line) a file name to be used by the program. The prefix is always returned in SOS file name format.

Getting the Pathname of the Program Currently Executing

The format for the Get Pathname procedure call is

```
GET_PATHNAME(PATHNAME)
```

where PATHNAME is a VAR parameter of type string. The variable passed to this procedure is set to the pathname of the currently executing

4

Using the Executing Program Pathname

4
Using the Executing Program Pathname

Pascal 1.1 provides a helpful extension to pathnames for use in application program development. You may now use the percent character (%) to mean "the same directory path as the executing program." For example, if the program

```
/PROFILE/APP1/SUB1/FOO.CODE
```

is currently being executed, the % character stands for the directory pathname

```
/PROFILE/APP1/SUB1
```

during the execution of this program and until another program is executed.

Instead of giving, for example, the complete pathnames of data files used by the program:

```
/PROFILE/APP1/SUB1/DATA1
/PROFILE/APP1/SUB1/DATA2
```

you can now simply call them by their local file names in your program:

```
%DATA1
%DATA2
```

This new pathname extension allows you to write an application program without knowing the specific path to the location of the local files. To use it, you first place the local files, such as the data files above, in the same directory as the executing program, and then you use the % character, whenever you need it, as a substitute for the directory pathname. This capability frees you from having to know and use the specific directory path of the program (and hence of its library files) when you are creating the Library Name File. This provides directory independence, so that you can move or copy an application program from one directory to another--from a flexible disk to a rigid disk device, for example--without changing the program. Specifically, it allows a programmer to permit the end user to place

When you execute a program, the % character is set as soon as the system has determined that the pathname of the program is valid and that the local code file (for example, FOO.CODE) exists. The symbol is not set to another pathname while the same program is executing, but when you execute another application program or a system program, such as the Filer, Editor, or Compiler, then the % is set to another directory pathname, which is that of the new program.

Although you can use the % character any time as a directory path--with the List command in the Filer, for example--note that it has three basic uses:

- naming files in a library name file;
- accessing files during program execution;
- chaining to other programs during execution.

Using the Percent Character in a Library Name File

Because the executing program pathname (%) is set as soon as the code file for the program has been found, you can use it in the Library Name File to replace the directory pathnames of the listed library files (see Chapter 2). If you had this set of files:

```
/PROFILE            {volume root directory name}
  /APP1             {subdirectory name}
    FOO.CODE        {an executable program}
    FOO.LIB         {a Library Name File}
    APP1.LIB        {a library file}
    APP2.LIB        {a library file}
```

and wanted to use the new pathname symbol, the contents of the Library Name File for FOO.CODE, which is FOO.LIB, would be

```
LIBRARY FILES:
%APP1.LIB
%APP2.LIB
$$
```

Then when you execute /PROFILE/APP1/FOO.CODE, the system opens up the Library Name File FOO.LIB and reads the pathnames for the two library files APP1.LIB and APP2.LIB. In this case the system expands the pathnames like this:

```
%APP1.LIB  -->  /PROFILE/APP1/APP1.LIB
%APP2.LIB  -->  /PROFILE/APP1/APP2.LIB
```

The % stands for the directory path /PROFILE/APP1 of the program

Keep in mind when developing an application that the grouping of related programs and their libraries together in the same directory allows you to use the % character to specify library files.

Accessing Files During Program Execution

Most application programs require the use of numerous files (like data files, output files, temporary files, and so forth) during execution. These files usually reside in the same directory as the main program. Using the % character, you can name these files in the main program without having to know their directory paths when the program is executing. For example, if the program

```
/PROFILE/APP1/GORN.CODE
```

uses the files DATA1 and DATA2, you would want to group the set of programs in the same directory:

```
/PROFILE              {volume root directory name}
  /APP1               {subdirectory name}
    GORN.CODE         {an executable program}
    DATA1             {a data file name}
    DATA2             {a data file name}
```

Then in the source code for program GORN.CODE, you can specify the two data files using these strings:

```
'%DATA1'
'%DATA2'
```

Here are two examples of source code showing possible uses of the strings now specified with the % character:

```
RESET(A_FILE, '%DATA1');
REWRITE(B_FILE, '%DATA2');
```

Thus you do not have to specify the actual directory path (in this case, /PROFILE/APP1). Whoever uses this program--you or someone else--is free to place this set of files in any directory, with any name, as long as they all reside in the same directory and as long as that directory is on line at the time of program execution.

Chaining to Other Programs During Execution

When a set of programs is to be chained together during execution, you can use the % character to specify the pathname of the next program to be linked and executed. For example, if you want the set of programs

```
/PROFILE            {volume root directory name}
  /APP1             {subdirectory name}
    FOO.CODE        {an executable program}
    BAZ.CODE        {an executable program}
    GORN.CODE       {an executable program}
```

to be executed in the order of FOO.CODE --> BAZ.CODE --> GORN.CODE, you use these pathnames in the call to the SETCHAIN procedure:

- In FOO.CODE use the procedure call

```
SETCHAIN('%BAZ');
```

- In BAZ.CODE use the procedure call

```
SETCHAIN('%GORN');
```

By using the % to specify the next file to be linked, you avoid having to state the complete pathname. To start the execution of the chain, you execute

```
/PROFILE/APP1/FOO
```

Again, all that is necessary is that you place the files on line and in the same directory.

For an explanation of the SETCHAIN procedure, see Appendix C in the Apple III Pascal Programmer's Manual, Volume 2.

5

Arranging Files on Disk: a Sample Application

5
Arranging Files on Disk: a Sample Application

The new features of Apple III Pascal discussed in the previous chapters--such as extended libraries and several ways of manipulating pathnames from within executing programs--were added specifically to help you in the development of applications. Now, in this chapter, you will see how to arrange your code and library files in directories on a flexible disk when making a runtime version of a complex application. Advanced programmers who understand the process of structuring an application may skip this chapter and go to the next.

Preliminaries: a Significant Development Stage

Planning how to arrange your files on disk is a significant application development stage, for the way you arrange your files will make a difference in the executability of your application.

Important Decisions to Make

As you plan file layout for the runtime disk, you'll want to make the following important decisions:

- Which library files will contain the intrinsic units required by your application?
- Do you intend any library files to function as shared library files in your application?
- In what directories and subdirectories should your code files and library files be grouped?
- Do you want to make a turnkey application by including a SYSTEM.STARTUP file?
- Will you need to document special instructions for the end user

- How many Library Name Files do you need to make, and which library files will each one point to?

- Have you included any necessary chaining commands in your programs?

- Do you need to make provision for any temporary files that the application will require in the course of running?

Getting Started: Your New Application

In the following sections you'll go through the actual steps of structuring the files of a particular application. Assume that you, the application developer, have completed the coding, compiling, and linking of the main programs and procedures making up this new application. And you know what intrinsic units the application will require at run time. Now, with the Pascal Filer and any other necessary utility programs at your fingertips, you are ready to copy all the relevant files to the disk or disks that will hold the runtime version.

Your hypothetical application is a data base manager (named SUPER MANAGER) with three parts, each developed as a main program:

- A top-level program that handles most of the user interface and that dispatches to two other programs that do most of the work.

  File name is DBM.CODE.

- A program that handles record definition and data entry.

  File name is ENTRY.CODE.

- A program that handles sorting and report generation.

  File name is REPORT.CODE.

Arranging the Intrinsic Units Your Application Requires

In order for the system to find the intrinsic units that your application programs require at run time, you need to identify them and place them in library files, if you've not done so already.

Identify the Intrinsic Units Needed

Your programs use several intrinsic units, some originally resident in

identified here with an asterisk (*). Table 5-1 indicates which program uses which intrinsic units.

Intrinsic Unit	Description	Used By
APPLESTUFF*	Procedures and functions like RANDOM and KEYPRESS.	DBM.CODE
CHAINSTUFF*	Three procedures to interrelate programs, like SETCHAIN.	DBM.CODE ENTRY.CODE REPORT.CODE SYSTEM.STAR.LIB
DBMUTILITY	Your own set of utility procedures.	ENTRY.CODE REPORT.CODE
PASCALIO*	Pascal input and output procedures, like SEEK.	ENTRY.CODE REPORT.CODE
SCREENMANAGER	Your procedures to manage screen display.	DBM.CODE ENTRY.CODE REPORT.CODE
SORTROUTINES	Your own sorting routines and utilities.	REPORT.CODE

Table 5-1. Intrinsic Units Used by SUPER MANAGER

Make Library Files to Hold the Intrinsic Units

You need library files to hold these units, files with descriptive names to help you identify the contents and to give the system access to the intrinsic units at run time. Probably you'll defer the actual making of these new library files until you're ready to move all the required files onto the final runtime disk, but assume that, looking ahead, you decide now to handle your library needs in the following manner.

First, you copy the units you need from SYSTEM.LIBRARY to their own files with, say, these names:

- APPLESTUFF unit moved to new file APPLE.LIB
- CHAINSTUFF unit moved to new file CHAIN.LIB; also to SYSTEM.STAR.LIB

Of course you could place these units together in one library file, rather than three separate ones.

Note that because the new library file names clearly identify the contents, you'll not lose track of where you've moved these units that you copied from the original SYSTEM.LIBRARY. The CHAINSTUFF unit is included because you have three main programs, as well as a SYSTEM.STARTUP file, that will have to chain together during execution of the application.

Second, if you haven't already collected your other intrinsic units into distinct library files, you'll want to do it soon:

- DBMUTILITY procedures collected in new file DBMUTIL.LIB
- SCREENMANAGER procedures collected in new file SCRMAN.LIB
- SORTROUTINES procedures collected in new file SORT.LIB

Good. You've identified all the library files your application will need. Shortly, you can place them in their proper directories on disk.

Preparing the Final Runtime Disk

Now you need to get a flexible disk ready to receive the code files and library files of your application.

Format the Disk and Give the Root Directory a Name

Format a new blank disk--or reformat an old one--using the System Utilities Format command. What do you want for the volume root directory name? How about /SUPERMAN?

Select a Startup Option for Your Application

You are planning a turnkey application, one that does not require a manual startup. If there were room for everything, you'd place the SOS startup files, the Pascal system files, and your code files and library files all on one disk so that the user would merely have to power up to get the program running. But because your code files and library files together use, say, about 200 blocks of disk space, you'll have to provide the next best option, namely a disk that includes everything except the SOS startup files. The user will then perform a two-stage startup using your SUPER MANAGER disk after using a separate SOS startup disk.

Now finish your disk preparation detail by copying these fil

```
SYSTEM.PASCAL       }    to volume root directory /SUPERMAN
SYSTEM.MISCINFO     }
```

But there are two more files to copy to your program system disk:

```
SYSTEM.STARTUP      }    to volume root directory /SUPERMAN
SYSTEM.STAR.LIB     }
```

At execution time, SYSTEM.STARTUP, using CHAINSTUFF, which you had placed in SYSTEM.STAR.LIB, will chain to DBM.CODE, the top-level program, and hence start up the application, provided SOS is already started up.

Copying Your Code Files and Library Files to the Runtime Disk

Now you can move the code files and library files of your application to the new disk, to give it the structure you want it to have in the final runtime version. As part of that process, you'll make a Library Name File to correspond to each main program code file. Because your application will go out to the end user on a single flexible disk (along with a SOS startup disk), all the files will reside in the same directory, the volume root directory /SUPERMAN.

Copy Your Main Program Code Files

First, copy your main program code files to the new disk:

```
DBM.CODE            }
ENTRY.CODE          }    to volume root directory /SUPERMAN
REPORT.CODE         }
```

Copy Your Intrinsic Units Into Library Files

Next, transfer your intrinsic units to the same volume and place them in the library files you planned above:

```
APPLE.LIB           }
CHAIN.LIB           }
PASCALIO.LIB        }    to volume root directory /SUPERMAN
DBMUTIL.LIB         }
SCRMAN.LIB          }
SORT.LIB            }
```

Make the Necessary Library Name Files

Now make three Library Name Files to correspond in name to the three main programs, substituting the suffix .LIB for .CODE. In each Library

by the program associated with that particular Library Name File, both the library files that only this program uses and those it shares with one or both of the other programs. This is how your Library Name Files will look:

```
DBM.LIB . . . . . {  LIBRARY FILES:
                     %APPLE.LIB
                     %SCRMAN.LIB
                     %CHAIN.LIB
                     $$              }

ENTRY.LIB . . . . . {  LIBRARY FILES:
                     %SCRMAN.LIB
                     %PASCALIO.LIB
                     %DBMUTIL.LIB
                     %CHAIN.LIB
                     $$              }

REPORT.LIB . . . .  {  LIBRARY FILES:
                     %SCRMAN.LIB
                     %PASCALIO.LIB
                     %DBMUTIL.LIB
                     %SORT.LIB
                     %CHAIN.LIB
                     $$              }
```

If you haven't done so already, place these Library Name Files on the new disk.

Check Your New Disk Directory

Now the directory of the new disk should include these files, which are necessary to start up and run your application when SOS is started up.

```
/SUPERMAN
   SYSTEM.PASCAL
   SYSTEM.MISCINFO
   SYSTEM.STARTUP
   SYSTEM.STAR.LIB
   DBM.CODE
   DBM.LIB
   ENTRY.CODE
   ENTRY.LIB
   REPORT.CODE
   REPORT.LIB
   APPLE.LIB
   CHAIN.LIB
   PASCALIO.LIB
```

Note that the following sequence of actions is necessary to execute the application. First the user starts up SOS. Then when SOS asks for the system disk containing SYSTEM.PASCAL, the user will insert the SUPER MANAGER disk in the system drive. At this point the application will start running.

Some Final Details to Consider

Before you run your application on the new disk to see how it performs, you need to give attention to a few minor but important matters. They have to do with chaining programs, opening temporary files, and running the application on large disk.

Chaining Programs

You designed SUPER MANAGER in such a way that your turnkey program, SYSTEM.STARTUP, interacts with DBM.CODE, DBM.CODE interacts with the other two main programs, ENTRY.CODE and REPORT.CODE, and they in turn interact with DBM.CODE. This is accomplished through chaining, where each of the four programs (including SYSTEM.STARTUP) declares "USES CHAINSTUFF" and where you place chaining statements in each program in the appropriate places. As we indicated in Chapter 4, you can use the % character to specify in a statement the directory pathname of the next program to be chained.

Thus, in DBM.CODE, you'll use the commands

```
SETCHAIN ('%ENTRY.CODE')
SETCHAIN ('%REPORT.CODE')
```

and in the programs ENTRY.CODE and REPORT.CODE, you'll use

```
SETCHAIN ('%DBM.CODE')
```

at the points where your application needs to go from one program to another. (Actually, you don't need to use the .CODE suffix in these commands.) Thus, when the program is executed, SYSTEM.STARTUP will chain to /SUPERMAN/DBM.CODE, and DBM.CODE and the other two programs will chain in turn as instructed.

This is a good time in the file structuring process to check to see that your program declarations and program code include the proper chaining instructions.

Opening Temporary Files

While executing, the program REPORT.CODE does some file sorting, which requires the creation of temporary files. The program can take care of creating those files as needed, but only if you make sure it includes a code line of the form

```
REWRITE (F, '%SORT1')
```

each time it needs to open or reopen a temporary sort file, where F is the identifier of a file variable (already declared) and the string value is the pathname of the file to be opened by the built-in procedure REWRITE. By using the % character in the string, you tell the system to supply the directory pathname used by the executing program. (See Chapter 4 if you want to review the use of this special extension. Use of the REWRITE procedure is explained in the Apple III Pascal Programmer's Manual, Volume 1.)

Running Your Application on a Rigid Disk Device

You can anticipate that some of the end users of your data base will want to run SUPER MANAGER on a rigid disk device like ProFile. In that case, you will want to include special instructions, perhaps near the beginning of your manual, for transferring the files to the rigid disk. You will direct the user to copy all the codefiles and library files from the application flexible disk--except the files SYSTEM.STARTUP and SYSTEM.STAR.LIB, which won't be needed--to a new subdirectory (called /SUPERMAN or whatever you like) on the rigid disk.

To execute the application, provided you have already started up SOS and powered up a ProFile disk drive, you will type X and the pathname

```
/PROFILE/SUPERMAN/DBM.CODE
```

Now SUPER MANAGER should run either on flexible disk or on rigid disk, ready for a final testing phase before production and release.

The chapters following this one tell you of many other ways that Pascal 1.1 will facilitate the development of applications, such as generating a program listing by request at compile time.

6

Using the ASCII/Binary Floating-Point Conversions

6

Using the ASCII/Binary Floating-Point Conversions

Two new conversion routines are available to you with Pascal 1.1. One converts an ASCII string of digits and an integer to the Pascal real number (binary floating-point) format. The second converts a Pascal real number (binary floating-point) to an ASCII string of digits and an integer. These new routines will allow you to build your own input and output routines instead of using Read and Write commands. The meaning of the ASCII string and the integer are indicated in the following examples:

1. An ASCII string, say, of 3452779 and an integer of -4 are interpreted as 3452779 * 1Ø^-4 = 345.2779.

2. An ASCII string, say, of -22491 and an integer of 3 are interpreted as -22491 * 1Ø^3 = -22491ØØØ.

These routines typically will be useful when you need to edit and convert a keyboard input to a Pascal real (binary floating-point) or, on the other hand, to write out a Pascal real (binary floating-point) as a text string. To do these, you may, of course, use the previously available text I/O procedures READLN and WRITELN. However, the latter are limited in the way they handle numerical conversions. If your design requires more control over such conversions, you can use the two new conversion routines. No USES statement is required in order to use these new routines. However, the PASCALIO unit must be available at runtime in an appropriate library file.

The ASCII to Binary Floating-Point Conversion

A new function in the PASCALIO library unit, STRTONUM, is defined as

```
FUNCTION STRTONUM (VAR DECSTR : STRING; POWEROFTEN : INTEGER) : REAL;
```

where DECSTR is any string of ASCII characters and POWEROFTEN an integer (a power of ten) to be converted together to a real number. STRTONUM returns the correctly rounded real. If the first character is a sign (+ or -), it is handled correctly. Otherwise, every character

the Power of Ten parameter (POWEROFTEN) to indicate where the decimal point should be in the converted number. For example:

```
If      S := '12345';
then    STRTONUM (S, -3) gives the real value 12.345
and     STRTONUM (S, 2) gives the real value 12345ØØ
```

The length of the string must be less than 29, and blanks are treated as zeros. If the length of the string is greater than 28, then STRTONUM generates a NaN as the result. Otherwise, POWEROFTEN can be any integer and is used to scale the value returned.

The Binary Floating-Point to ASCII Conversion

A new procedure in the PASCALIO library unit, NUMTOSTR, is defined as

```
PROCEDURE NUMTOSTR (R :  REAL; FIXED :  BOOLEAN; PLACECOUNT :
                    INTEGER; VAR S :  STRING; VAR EXPON :  INTEGER);
```

where R is a real number to be converted to a string of ASCII characters and an integer, where FIXED specifies a fixed-point or a floating-point output format, and where PLACECOUNT gives either the total number of digits (in floating-point format) or the number of places to the right of the (assumed) decimal point (in fixed point format).

Note that the digit string never contains a decimal point or a leading zero; final output format must be constructed from S and EXPON. For example:

```
NUMTOSTR (Ø.ØØØ1234, FALSE, 4, S, EXPON)
```

will set S to '1234' and EXPON to -7. You can now use these values to format the number for output as you wish.

If S is too short to hold the output string, a string assignment error will occur. Other errors--like attempting to format an infinity or NaN, or specifying PLACECOUNT <1 when FIXED is FALSE, or specifying PLACECOUNT <Ø when FIXED is TRUE--will return S = '***' with EXPON unchanged.

If you are not familiar with the Pascal type REAL and the way the Pascal compiler handles it, please consult Chapter 3 in the Apple III Pascal Programmer's Manual, Volume 1. For more information on the single-precision floating-point numerics standard followed here, see Appendix E in Volume 2 of the same manual.

7

Listing a Program at Compile Time

7
Listing a Program at Compile Time

The Pascal Compiler now gives you two ways to generate a program listing: you may use the LIST option previously available or you may use the Listing File option at the time you compile your program. For more information on the LIST option, see Appendix F in the Apple III Pascal Programmer's Manual, Volume 2.

The second option allows you to request a program listing immediately before your source program is compiled. As soon as your program calls the Compiler, you are asked for the source file name, the code file name, and the file name where you want the listing to be placed. Here is the request line the Compiler presents to you:

```
Listing file (<ret> for none or option in source):
```

Now you are given three options:

- Press RETURN. If your program already contains a LIST option that requests a listing and specifies a listing file, then the listing will be made. Any L+ or L- option will be carried out. If there is no LIST request in the source program, then no program listing at all will be made.

- Type a pathname. This action causes the listing to go to the file you name here, overriding any LIST option given in the source. L+ and L- are still usable.

- Press ESCAPE, then RETURN. This action cancels compiling and returns you to the command level.

Thus you may choose the LIST option at the time you write the source code, and in addition you may either confirm that option at the time you compile or use the Listing File option to make a new listing request.

8

Making a Rigid Disk Your System Disk

8
Making a Rigid Disk Your System Disk

Your system will run faster if you transfer your Pascal system files to a rigid-disk device, such as the Apple ProFile. Pascal 1.1 has a program that allows you to move the Pascal system to any blocked device, including a rigid disk. This program--called PMOVE--takes a first-stage Pascal startup disk and changes SOS.INTERP so that the device you specify will become the Pascal system device, identified by the system as unit #4. As a result, the files SYSTEM.PASCAL, SYSTEM.MISCINFO, and SYSTEM.LIBRARY can now operate from a rigid disk instead of from the built-in drive of the Apple III. The remaining Pascal files, such as SYSTEM.COMPILER and SYSTEM.EDITOR, can optionally be placed on the rigid disk for faster execution.

Using PMOVE

Before beginning the PMOVE procedure, check the following:

1. Make sure that your rigid system disk is connected to your Apple III and powered up.

2. Using your System Utilities program, check the SOS.DRIVER file on your current boot disk to be sure it contains an active driver for the rigid-disk device you want to be the new Pascal system device.

   Note: If you are going to be moving the Pascal system files to a ProFile, you can use the ProFile driver already included in the SOS.DRIVER file on your new PASCAL3 disk.

Once you have made these preparations, you are ready to begin the PMOVE procedure. Follow these steps:

1. Place your PASCAL1 Version 1.1 disk in the built-in drive and start up the system by turning on the power switch or pressing CONTROL-RESET.

2. Arrange the SOS files and the Pascal system files as follows:

   Onto a new Pascal startup flexible disk, formatted for Apple III

   Copy SOS.KERNEL from your PASCAL1 disk (Version 1.1)
   Copy SOS.INTERP from your PASCAL1 disk (Version 1.1)
   Copy SOS.DRIVER from your PASCAL3 disk (Version 1.1)

   Note: Actually, you may copy your most recently configured SOS.DRIVER file, as long as you make sure it contains a driver for your rigid system disk.

   Onto your rigid system disk (at volume root directory level)

   Copy SYSTEM.PASCAL from your PASCAL1 disk (Version 1.1)
   Copy SYSTEM.MISCINFO from your PASCAL1 disk (Version 1.1)
   Copy SYSTEM.LIBRARY from your PASCAL3 disk (Version 1.1)

   Note: You should copy the expanded version of SYSTEM.LIBRARY found on the PASCAL3 disk, not the one on the PASCAL1 disk. You may optionally copy any of the remaining Pascal files from your new Pascal 1.1 disks to your rigid system disk.

3. Place your PASCAL3 Version 1.1 disk in an external flexible disk drive. (PMOVE is on PASCAL3.)

4. At the Command line, type X for execute, and then type PMOVE.

5. When the program PMOVE asks for the SOS device name of the new system device, enter the name of your rigid system disk in SOS format (.PROFILE, for example).

6. When the program PMOVE tells you to insert the disk containing SOS.INTERP and enter the SOS device name of the drive, insert the new Pascal startup disk (the one you made in step 3 above) into a drive and type in the name of the drive in SOS format (.D2, for example).

   Note: If you have no free drives, you may remove your PASCAL3 disk and place your new startup disk in the now empty drive.

7. When the program terminates successfully, restart the system using the new startup disk. You can remove the startup disk after this. The Pascal system should now be running on the rigid disk. You will know that your system is working correctly if the Apple III Pascal prompt line comes up on the screen. If it doesn't appear, check the above steps in order

Comparing Versions of SYSTEM.LIBRARY and SOS.DRIVER Files

For your convenience, here are the contents of the two different SYSTEM.LIBRARY files and the two different SOS.DRIVER files:

SYSTEM.LIBRARY on PASCAL3 contains

```
APPLESTUFF
CHAINUNIT
LONGINTIO
PASCALIO
PGRAF
TURTLEGRAPHICS
REALMODES
TRANSCEND
SANE
ELEMS
```

SYSTEM.LIBRARY on PASCAL1 contains

```
APPLESTUFF
CHAINUNIT
LONGINTIO
PASCALIO
```

SOS.DRIVER on PASCAL3 contains

```
.GRAFIX
.SILENTYPE
.AUDIO
.PRINTER
.CONSOLE
.PROFILE  {configured for slot #4}
+.FMTD1
+.FMTD2   {disk format drivers to be used with the}
+.FTMD3   {Utility Filer, which may be placed on  }
+.FTMD4   {the ProFile rigid disk.                 }
```

SOS.DRIVER on PASCAL1 contains

```
.GRAFIX
.SILENTYPE
.AUDIO
.PRINTER
.CONSOLE
```

9

Other Features of Pascal 1.1

9
Other Features of Pascal 1.1

In addition to the features described in the previous chapters, Pascal 1.1 contains the improvements below.

A Change to the Editor

Under the new version of Pascal, the Editor will correctly read a file with control characters in it. These characters will be displayed on the console in the current system font, which has a mnemonic symbol for each control character. Thus you can edit a file that has control characters.

The Screen Display in Program Chaining

Version 1.1 of Pascal changes the way the system handles the console display when one program chains to another. In the 1.Ø version, the system reset the console to its default values at the completion of a program. Now when a program chains to another program, the console display is not reset by the system; the display remains in the same state after it completes a program. This allows you to chain programs that rely on the same display format and content, and it provides compatibility with Apple II Pascal.

AIIFORMAT Has a New Look

The AIIFORMAT program on your PASCAL3 disk enables you to format disks that will run on an Apple II using Apple II Pascal or on an Apple III using Apple III Pascal. This improved version of AIIFORMAT has a better interface, giving you step-by-step instructions for a quick, effective format. Simply go to the Command level (first making sure that PASCAL3 is in a disk drive) and execute /PASCAL3/AIIFORMAT, letting the program take you through the format procedure.

A New System Level Command

A new command option has been added at the system Command level. This command allows you to exit the Pascal system and then start up another application, such as Visicalc III or Applewriter III. The new command is Q(uit. When you type Q at the Command level, the system will ask you whether or not you wish to leave the Pascal system. Type Y to exit the system.

A Modification of the Compiler

The compiler and linker have been modified to support up to 254 procedures in a single compiled unit. Previously, you could have only 143 procedures in your source file. Also, the compiler now accepts larger procedures by increasing the allowable size of the code generated for a procedure.